Lecture 13: Quantization and compression

DANIEL WELLER
THURSDAY, FEBRUARY 28, 2019
Agenda

Quantization and compression
Orthogonal representations and multiplexing
Compressive transforms: cosine, wavelet
JPEG compression

These images are nearly indistinguishable from each other. However, the image on the right is JPEG-compressed to require only a single bit per pixel on average, a significant reduction in memory over the 8 bits normally used to store 256 levels in a gray image.
How do we measure compression?

**Compression Measures**
- **Bits Per Pixel (BPP)** is the average number of bits required to store a value for a pixel in an image (the average code word length for a signal).
  - In an non-coded image, \( BPP = \log_2(K) = B \), where \( K \) = the # of allowable levels. Usually \( B = \log_2(256) = 8 \) for gray images. The number of bits used to code pixels may vary across a coded image.
  - Let \( B(i, j) = \# \) of bits used to code pixel \( I(i, j) \). Then \( BPP = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} B(i, j) \) for an image with \( N \times N \) pixels.
  - If the total number of bits contained in the code is \( B_{total} \), then \( BPP = \frac{B_{total}}{N^2} \)
- **Compression ratio (CR)** is the ratio \( CR = (B/BPP) > 1 \)
- Both BPP and CR are used frequently.
Quantization as we described earlier refers to the process of assigning discrete values to a sampled signal, converting analog to digital. We discussed a few ways of quantizing numbers:

- Unipolar quantizers for unsigned (nonnegative) numbers
- Bipolar quantizers for signed numbers, using signed magnitude, two’s complement, offset binary
- Rounding and truncation
- Fixed and floating point (for decimals)

More generally, quantization assigns a sequence of bits to a given symbol (value). So quantization is really compression.

- For instance, Huffman coding is a way of compressing signals according to their symbols’ probabilities.
Quantization as compression

As such, quantization is a fundamental aspect of coding theory, which studies different ways of compressing and representing information efficiently.

Some elements of quantization more general than before:

- Symbols can contain non-numeric information (e.g., “A”, “C”, “G”, “T” in DNA)
- Symbols can contain multiple “letters”, allowing what we call vector quantization
- Quantized values are not necessarily ordered, and resulting bit strings do not have to be the same length (variable-length codes)
- Multiple symbols can get quantized to the same code word. This is common in lossy compression. (Recall: lossy compression may have nonzero error in the decoded signal.)
Scalar quantization

Start with a set of possible symbols $S$ (suppose there are $K$ of them).

Assign each symbol to one of $M$ codes. Note that this is necessarily lossy compression if $M < K$.
- The chosen code is “closest” to the symbol in some sense.

- *Scalar* quantization maps each symbol individually.

- Sources of compression: bits assigned to each code may be variable-length, there may be fewer codes than symbols.
Uniform scalar quantization

*Uniform* scalar quantization usually produces a “linear” set of quantized codes for a set of symbols. This is like the quantizer we’ve studied before.

For instance, quantize values between 0-99 by truncating down to the nearest multiple of 10:
- Quantized codes represent values 0, 10, 20, ..., 90.
- The symbol for 55 gets quantized as 50.

Another example: quantize values between 0 and 20 by rounding to the nearest multiple of 5:
- Quantized codes represent 0, 5, 10, 15, 20
- The symbol for 13 gets quantized as 15.
Nonuniform scalar quantization

If some range of values are more likely than another, choose the quantization codes more densely over that region. This quantization is *nonuniform*.

Example: If values from 0-19 are more likely than the others, perhaps use quantized codes for 0, 5, 10, 15 in that range and greater spacing for codes 20, 40, 60, 80.

Logarithmic spacing can accommodate a high dynamic range and is useful for such quantities.

For a lossy quantizer with a given number of codes, nonuniform quantization can reduce the distortion or error without increasing the number of bits.
Example

Let’s quantize nonuniformly with 8 levels spaced logarithmically from $2^{-7}$ to $2^0$. We’ll attempt to quantize values $1/n$ for $n = 1, 2, 3, \ldots, 100$ by rounding to the nearest level.

The quantized values are $1, 1/2, 1/4, 1/4, 1/4, 1/8, 1/8, \ldots$

The average error is 0.0065, and we only needed three bits per symbol (fixed code length).

What about a uniform scalar quantizer between 0 and 1 (rounding to 8 levels $1/16, 3/16, \ldots$)? This quantizer’s average error would be 0.0396, six times greater than for the nonuniform one!
Vector quantization

When signal values are correlated in some way, so the combined information is less than the that of each separately, vector quantization can reduce the quantization error further.

Vector quantization assigns codes to vectors or groups of values, usually in a nonuniform way.

The Lempel-Ziv method can be viewed as vector quantization.

Example of correlated information: height and weight of children. Each dot represents a vector-quantized code.

Credit: Guy Blelloch, “Introduction to Data Compression”, 2013.
Transform coding

The gains from vector quantization are due to the fact that the higher dimensional signal can be projected to a one-dimensional set by exploiting the correlations to eliminate redundancy.

More generally, we can *transform* a signal before quantizing it to achieve significant savings in bit rate.

This method is called *transform coding* and leads us to introduce the idea of orthogonal representations of signals.
Orthogonality

Suppose we are given a pair of signals $x[n], y[n]$. They are orthogonal to each other if their inner product equals zero:

$$\sum_n x[n]y[n]$$

We’ll talk a lot more about inner products later in the course. For now, we just define the formula above.

Some examples of orthogonal signals:

- Sine and cosine function (same frequency).
- A binary sequence (0 and 1) and its complement.
Orthogonal representations

What if we have a set of orthogonal signals $x_1[n]$, $x_2[n]$, ..., $x_N[n]$ and take a linear combination of them to form a new signal $y[n]$?

- We can extract the “component” $x_i[n]$ of the signal $y[n]$ by taking the inner product of $y[n]$ with $x_i[n]$, which will tell us how much of the signal is composed of $x_i[n]$.

- In general, we cannot assume a real signal has been constructed this way, but given a set of orthogonal components, it may still interest us to see how much signal content corresponds to each component. This is called the orthogonal representation of the signal.

- It turns out we are already familiar with one orthogonal representation: the frequency spectrum
  - The spectrum tells us how much of a complex sinusoid with a given frequency is contained in a signal.
  - This is very useful, but there are other useful orthogonal representations.
Common transforms

In addition to the Fourier transform we mentioned previously, several other transforms yield useful orthogonal representations for analysis or compression:

- Cosine or sine transform for signals composed of sinusoids
- Wavelet transform for signals composed of various sized simple “objects” (like shapes in an image)
- Karhunen–Loève transform for random signals

Question: what makes a transform useful for compression?
Transform coding example

Cameraman.tif (in MATLAB)

Original

Quantized

4-bit quantization (32 KB)

2D DCT

Quantize “analysis” coefficients differently

2D DWT

JPEG compression (~11 KB)
The cosine transform

The cosine and sine transforms are similar to the Fourier transform in that they measure signal components by frequency.

However, they use only sine or cosine functions. How does this work?

- Solution: for the cosine function, make signal an even function. For the sine, make it odd.
The cosine transform

Given an even-valued signal $x_e[n]$, we can construct a series of cosines at increasing frequencies to represent it:

$$x_e[n] \approx \sum_k a_k \cos \left( \frac{2\pi kn}{N} \right)$$

$N$ is the length of the even-valued signal (after reflecting).

Here, the values $a_k$ are “frequency spectrum” values for the frequencies $k/N$, $k = 0, 1, 2, \ldots$. Unlike the Fourier frequency spectrum we’ve discussed previously, these $a_k$’s are real-valued for real-valued signals.

The discrete sine transform does the same with an odd-valued signal $x_o[n]$, using sine functions of increasing frequency.
A note about different cosine transforms

There are many cosine transforms. Why different ones?

- We can choose whether to reuse coefficients on the boundary of the signal/image.
- Example:
Why the cosine transform?

We already observed that many signals can be described in terms of their frequency content.
  ◦ Audio: musical notes, speech sounds
  ◦ Images, videos, etc.

The cosine and sine transforms have the added advantage of keeping things real-valued (no need to keep track of real and imaginary parts of complex numbers).

Many real signals have limited frequency content, and much of the content are at relatively low frequencies.
  ◦ Thus, the cosine transform has an “energy compaction” property, as in most of the signal is described by just a few coefficients of the spectrum.
Example of energy compaction

The cosine transform (DCT-2) does even better than the Fourier transform (DFT).
Cosine transform

Let’s see what happens if we take our cameraman and compute a cosine transform for the whole image:
Cosine transform

So it appears most of the coefficients are zero. Why is this procedure not a good idea for accurate compression?

- The coefficients are actually just very small, relative to those in the top-left corner. It would be hard to match them all due to the dynamic range.

- Solution: divide the image into small “blocks”, and transform each block individually.
Wavelets: another transform

The wavelet is another way of describing an image or signal in terms of orthogonal components. It combines two ideas: separate a signal into “low” and “high” frequencies, rescale the low-frequency signal and transform again.

- This process is not affected by the cosine transform scaling issue, so we can apply to the entire image.
Wavelets

There are many different kinds of wavelets, distinguished by the pair of low- and high-frequency filters they use. Some are orthogonal:

- Haar wavelet
- Daubechies wavelet

MATLAB contains GUI software for constructing Wavelet transforms if you install the Wavelet Toolbox. Look for the “Wavelet Design & Analysis” button under the Apps toolbar in MATLAB if you’ve installed it. To the right is the Wavelet 2-D tool after loading the “cameraman” image and clicking “Analyze”.

JPEG compression

JPEG is usually a lossy compression format for images that divides an image into small rectangular blocks, employs transform coding on each block, and performs Huffman coding on the resulting quantized binary representation of that transform.

For color images, the luminosity (gray) and color components are usually compressed separately.

MPEG is a video compression that uses similar ideas but also tracks changes between video frames so not every frame has to be stored completely.
JPEG compression step-by-step

1. Convert RGB to luminance, chrominance (usually).
2. Divide image into minimal blocks (8x8 or 16x16) for cosine transform (not needed for wavelet).
3. Use 2D discrete cosine transform on each block.
4. Quantize the coefficients individually, using more bits for the lower frequency components, and fewer bits for the higher frequency components.
5. Huffman code the binary representation. Store the Huffman code tree and compressed bits.
JPEG compression result

Bit rate vs. average error for Cameraman JPEG Compression

Average Rate (bits/pixel) vs. Average Error (dB)
JPEG compression result

Significant reduction in bit rate (bits per pixel) is possible with a minimal change in image quality.

Even with very high compression, the errors mainly show up in smooth background areas.

An automatic JPEG implementation would determine the minimum rate required to achieve a maximum allowable perceptible error. Most image processing software let you specify a “quality number” between 0 and 100 to control this (75 is the default in MATLAB).
Announcements

Next time: Information and communication